Universidade de São Paulo

Escola Politécnica, Engenharia de Computação
PCS3732 - Laboratório de Processadores
Bruno Abrantes Basseto & Marco Tulio Carvalho de Andrade

PiFork

Relatório de projeto

Nome Completo	**N USP**
Henrique Freire da Silva	12555551
João Pedro Zangelmi Rezende de Menezes	11222822
Nathan Filipe Moreira dos Santos	12566891

São Paulo, 16 de Agosto de 2024

1 CONTEXTO

Em Sistemas Operacionais (SOs), a administração de recursos e do ambiente de execução são abstraídos dos usuários hóspedes. Isso se dá através de diversos componentes que constituem o sistema e organizam a interface entre hardware e software, englobando o gerenciamento de: processos; memórias; arquivos; e dispositivos.

A superfície de interação do usuário é delimitada através de uma camada de chamadas de sistema definida pelo SO, tal que suas ações dentro do sistema computacional sejam consideradas conforme permissões de acesso neste.

Essas chamadas de sistema podem ser agrupadas segundo o contexto de aplicação. Na especificação UNIX, é possível ressaltar aquelas relativas ao gerenciamento de arquivos (e.g. *open, read, write, close, lseek, stat/fstat*) e diretórios (e.g. *opendir, closedir, readdir, mkdir, rmdir*), de rede (e.g. *socket, bind, listen, accept, close, send, recv*); à coordenação de memória (e.g. *brk/sbrk, mmap*); e à administração de processos (e.g. *getpid, fork, wait/waitpid, exit, exec*).

Cada processo de usuário possui metadados associados e informações sobre o estado atual deste no sistema. Com a devida aplicação das chamadas mencionadas, este processo pode obter propriedade sobre recursos compartilhados no sistema e, inclusive, gerar novos processos com um mesmo (ou outro) contexto de trabalho. Neste sentido, a chamada *fork* pode ser utilizada para clonar o processo chamador e seu contexto a um novo ingressante do sistema, o qual terá um novo identificador e será associado ao original em hierarquia. A Figura 1 apresenta um diagrama do comportamento do serviço fornecido pelo sistema.

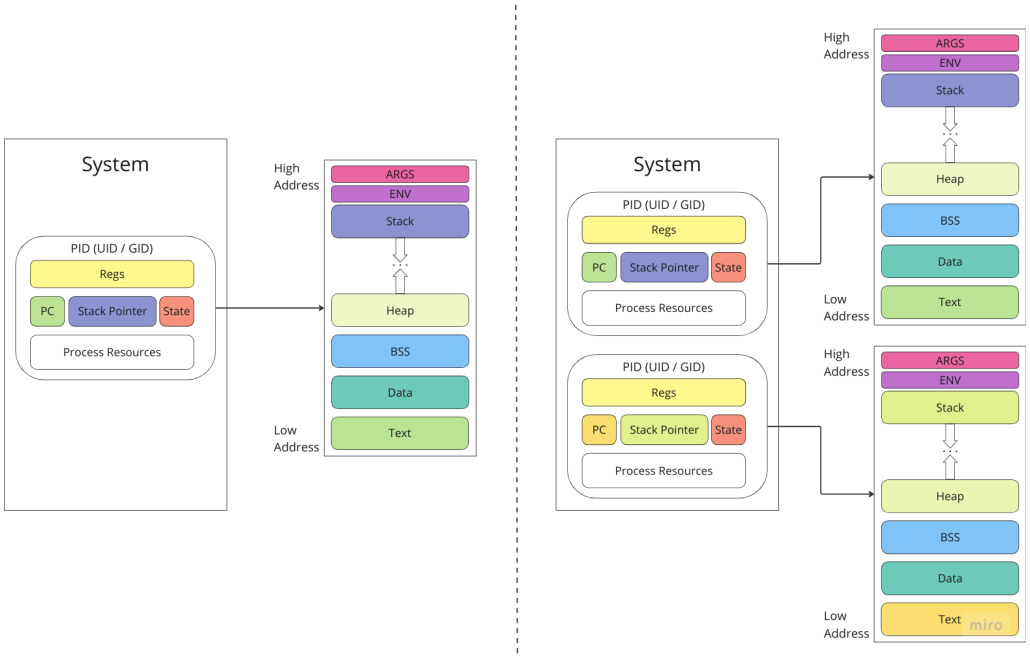

Figura 1: Representação simplificada da chamada de sistema (*syscall*) *fork* em um sistema operacional. (Fonte: autoria própria)

O estado do sistema é separado no antes (à esquerda da linha tracejada) da execução do *fork* e o depois (à direita). Percebe-se a introdução de um novo processo onde era contabilizado apenas um, junto a uma nova região de memória (ilustrada em segmentos, porém se estendendo a partições e

páginas) alocada ao usuário em questão. Pelo esquema de cores na Figura 1, infere-se a distinção dos processos quanto à etapa de execução - apontada pelo *Program Counter* (PC) de cada - e a referência de dados a qual cada recorre - inicialmente, ambos tem os mesmos dados (indicado pela mesma cor nos segmentos de *heap* dos dois processos), porém em endereços **físicos** diferentes (como mostra os ponteiros das *stacks*).

O novo processo gerado pelo *fork* possui também um identificador (PID) diferente, porém pertencendo a um mesmo grupo (GID) e a um mesmo usuário (UID); a relação pai-filho é estabelecida entre os dois processos, o que permite a interação adicional por demais chamadas de sistema (e.g. *wait/waitpid*).

O presente relatório descreve o projeto que constitui uma prévia de um sistema operacional básico, implementando a administração de processos e proteção de acessos a um modelo de sistema executivo com escalonamento justo (*round-robin*) de processos. O sistema é embarcado em uma placa Raspberry Pi 2 e tem como foco as chamadas *fork*, *wait/waitpid* e *exit*, recebendo o nome de $PiFork$.

2 MOTIVAÇÃO

A combinação das chamadas de sistema é o que permite o acesso de funcionalidades não necessariamente conhecidas pelo usuário, aumentando a integração de programas em nível de usuário a famílias de sistemas operacionais - adequados a uma especificação uniforme - e controlando o acesso ao hardware hospedeiro à medida que centraliza o meio de comunicação.

O desenvolvimento de um subconjunto coerente dessas chamadas permite o teste de sua funcionalidade individual e síncrona em prol do entendimento, ainda que limitado, de um SO comercial e das dificuldades acerca deste.

Nesta linha, a escolha de implantar um sistema administrador de processos implica, sobretudo, em algum mecanismo de remapeamento de memória: um *loader* relocador (ou dinâmico) ou tradução de espaços de endereçamento. A primeira solução implica na adição de uma etapa extra na criação de processos, adicionando latência à chamada e não sendo ideal para frequentes trocas de contexto. Dessa forma, optou-se pela segunda opção através da componente MMU (*Memory Management Unit*) já incluído no chip do processador, de forma a explorar a especificação da microarquitetura ARMv7 para além da ISA e fomentar o desenvolvimento de programas a nível de sistema (*bare-metal*).

3 OBJETIVOS

3.1 Objetivo Geral

Desenvolver um sistema operacional reduzido, com gerenciamento de processos em memória e proteção inter-processos, e embarcado em uma placa de desenvolvimento sem auxílio de *firmware* ou *software* além de iniciadores de sistema (*bootloader*) próprios da fabricante.

3.2 Objetivos Específicos

- Integrar a MMU ao sistema e testar seus diferentes modos de operação;
- Discernir mapeamentos coincidentes na TLB (*Translation Lookaside Buffer*) por PID do processo proprietário;
- Aplicar políticas de acesso a seções arbitrárias de memória;

- Estruturar e documentar um sistema modular e estendível.

4 MODELO DE SOLUÇÃO

O sistema proposto fora implementado integrando C e *assembly* ARM[1]. A estrutura do projeto inclui três principais subsistemas: o escalonador (*scheduler*); o gerenciamento de memória virtual (através da MMU e TLB); e o gerenciador de tarefas. Esses subsistemas correspondem aos diretórios *sched*, *mmu* e *sys* dentro de *src*, respectivamente.

Como um modelo simplificado, o sistema executivo implementado se limita a um núcleo apenas, tal que todo o trabalho percebido como simultâneo seja, na realidade, uma concorrência com frequentes intercalamentos.

As chamadas de sistema são fornecidas como funções acessíveis a usuários do sistema que utilizar o projeto como base. Elas invocam o modo supervisor do processador com o argumento adequado para tratamento pelo gerenciador de tarefas, o qual executa o serviço correspondente com o processo atualmente ocupando o processador ou outro relacionado (a depender da chamada feita).

A interface providenciada a usuários é como segue o Apêndice A, em que se tem funções *staticinline* definidas diretamente em um arquivo de cabeçalho. Essa escolha é proposital e força uma cópia única do conteúdo de cada uma em seu ponto de chamada. Isso se deve a alguns fatores:

1. Como há proteção de seções de memória, deve-se garantir que o usuário terá acesso direto às chamadas, i.e. as funções devem estar sempre em uma região mapeada, executável e não privilegiada;
2. Toda e qualquer mudança na implementação das chamadas (e.g. reordenação ou conformação de parâmetros enviados ou recebidos para a chamada de supervisor) força uma recompilação de arquivos que incluam esse cabeçalho;
3. Assim como o atributo *naked*, um função que sempre posta "em linha"(*inline*) não produz preâmbulo ou prólogo;
4. O número de instruções inseridas não difere muito do que seria produzido como para preparar uma chamada de função convencional, assim reduzindo quaisquer penalidades de desempenho quanto velocidade de execução ou armazenamento em que essa troca poderia acarretar em outra ocasião.

Nas seguintes seções, os subsistemas são tratados de forma individual e com maiores detalhes.

4.1 Gerenciamento de memória

A MMU é uma unidade que permite a tradução de um espaço de endereçamento lógico a outro, em geral físico. A Figura 2 apresenta como uma tabela em memória é indexada para acessar o registro de endereço (e atributos) correspondentes à seção consultada.

O registrador $TTBR$ (*Translation Table Base Register*) contém o endereço inicial da tabela armazenada em memória, a partir do qual indexará os registros de tradução. Como a memória principal tem $4GiB$ e optou-se pela configuração da MMU de seções (definida como regiões contíguas de $1MiB$ de comprimento), tem-se um total de 4096 partições iguais da memória original, das quais tem-se aproximadamente 1/4 útil para usuários, visto que o *kernel* ocupa uma seção e parte do endereçamento mapeia periféricos da placa.

[1]Repositório disponível em: `https://github.com/H-Freire/pifork`

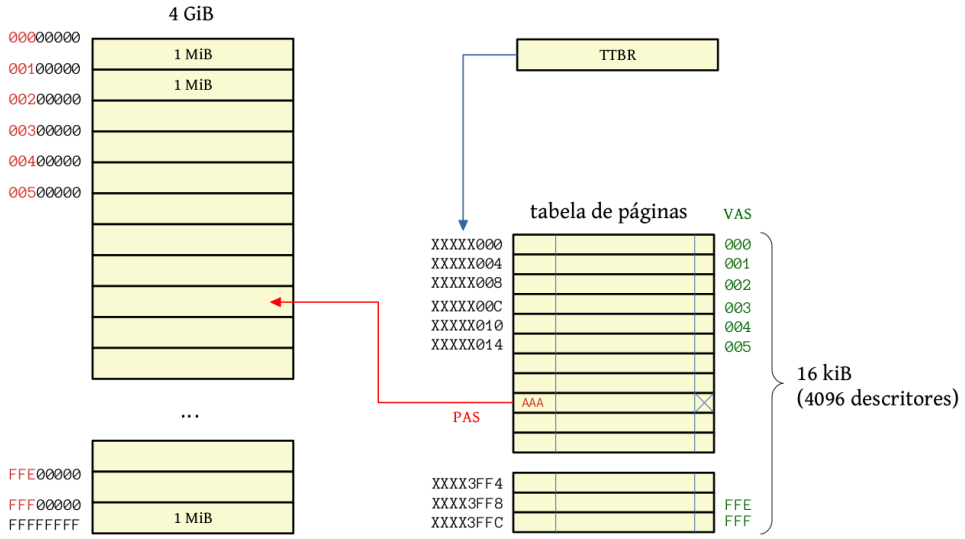

Figura 2: Tradução de endereço por página de $1MiB$. (Fonte: Bruno Basseto)

Algumas funcionalidades da MMU são abstraídas da movimentação de registradores do coprocessador de controle do sistema[2] para compor ações completas: inicialização e mapeamentos plano e por seção.

No cabeçalho interno, há a definição dos principais valores que podem atribuir características próprias dos registros de cada seção, tal como a possibilidade de cache (*write-through*, *write-back* ou nunca); acesso privilegiado, apenas para leitura não privilegiada ou acesso completo; não executável; e não global (própria do processo). Essas definições estão de acordo com a documentação disponível do ARMv7-A[3].

Quando o sistema é iniciado, a MMU se encontra desabilitada e a tabela de tradução deve ser preenchida antes que possa se habilitar esse componente, de forma a evitar exceções de *Page Faults* (faltas de página) no modo *Abort* do processador. Realiza-se então o mapeamento $1:1$ de endereços virtuais para físicos, porém aplicando separação lógica de seções pelo conteúdo que se espera situar nelas: a primeira seção (do *kernel*) recebe acesso privilegiado; seções a partir do endereço 0x3ff00000 recebem o atributo de "não executáveis", porém com permissão de escrita e leitura livre - comportamento associado aos periféricos como um todo, mas que poderia ser especializados a menos seções; e as seções de usuário, incialmente com permissões de leitura/escrita livres.

4.2 Gerenciamento de processos

O gerenciamento de processos se dá como a delegação das chamadas de supervisor aos devidos serviços internos. Além disso, esse subsistema mantém registro das seções de memória **física** ocupadas ao manter um vetor de bytes em memória, cada bit representando uma seção (1 ocupada, 0 desocupada).

Quanto às chamadas de sistema focadas no projeto, a implementação envolve o acesso a espaços de

[2]https://developer.arm.com/documentation/den0013/d/ARM-Processor-Modes-and-Registers/Registers/Coprocessor-15

[3]https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Virtual-Memory-System-Architecture–VMSA-/Translation-tables/Translation-table-entry-formats

endereçamento diferentes sem a perda de coerência da TLB, visto que um dos objetivos de projeto era a sua não invalidação (utilizando ASID para a indentificação de processos). Quando há a troca de contextos, o registrador $CONTEXTIDR$, do coprocessador 15, recebe o PID do processo ativo, de forma que sucessívels acessos sejam associados a este. Para manter a coerência dos dados e instruções, como se trata de um efeito colateral aplicado ao processador, deve-se aplicar métodos de sincronização de dados (DSB) e de instrução (ISB - *flush* nas instruções do *pre-fetch*).

O *fork* procura a primeira página física de memória livre, desabilita a MMU, realiza a cópia de todos os $1MiB$ do processo original (conforme o programa em *assembly* em B), reinicia a MMU e injeta um novo processo no sistema que corresponda à nova seção em memória. A MMU deve ser desabilitada durante a cópia já que não se pode assumir nada sobre a posição ou mapeamento em memória da seção livre escolhida; alternativamente, poderia se optar por invalidar qualquer mapeamento da seção livre encontrada.

O *waitpid* apenas bloqueia o processo atual ao alterar o seu estado. Caso seja fornecido um PID negativo do processo pelo qual esperar, entende-se que qualquer filho pode desbloqueá-lo; para um PID positivo, ao invés de colocar o processo pai em um estado de bloqueio padrão (que tem valor de inteiro -1), se utiliza do sistema de fraca tipagem da linguagem C para armazenar o resultado de $-(PID + 2)$, tal que possa se inferir qual o PID esperado apenas se avaliando o valor do estado de bloqueio do processo, mantendo os estados de bloqueio sempre negativos e permitindo PIDs positivos e nulos (0).

A chamada *wait* remapeia a um *waitpid* com o primeiro argumento igual a -1. Tanto o *wait* quanto o *waitpid* tiveram o último argumento (de opções) definido na especificação UNIX[4] removido por simplicidade.

O *exit* remove o processo que o chama do sistema. Adicionalmente, ele verifica se seu pai (processo imediatamente na hierarquia instituída) está bloqueado e se ele (filho) é um candidato válido para a suspensão do bloqueio. Caso seja, escreve seu status de saída em um ponteiro fornecido pelo pai depois de remapear a memória deste. O processo pai é então colocado em estado de pronto.

Para visão completa das chamadas de sistema implementadas, refira ao repositório do projeto.

5 TESTES

Para abranger melhor a proposta do projeto e provar a funcionalidade das chamadas de sistema, implementou-se 3 programas de exemplo que simulam uma estrutura comumente utilizada em SOs UNIX convencionais. Os exemplos se encontram no repositório do projeto, na pasta *examples* e no Apêndice C

Como se trata de um sistema *bare-metal* sem demais programas de suporte, utilizou-se de LEDs ligados a pinos GPIO da placa para apurar o funcionamento e estado do sistema e dos processos nele atuantes.

O programa *wait* (C.1) pisca 3 LEDs, cada qual de um processo diferente (um dos quais é gerado pelo *fork*), verifica a espera de um processo pai por qualquer um de seus filhos (no caso, há apenas um) e utiliza o PID desse filho que saiu do sistema para validar se é o correto e se sua saída foi com êxito, assim piscando um quarto LED de forma a sinalizar a chegada naquela seção de código.

O programa *waitpid* (C.2) executa o *fork* em *loop*, de forma a gerar múltiplos processos em uma hierarquia de largura 1 e profundidade 4. Cada processo tem um valor de pinagem em sua área de memória e utiliza tal valor para selecionar qual LED piscar; o processo original aguarda o seu

[4]Segundo parâmetro da chamada definida pelo UNIX: `https://linux.die.net/man/2/waitpid`

último filho gerado finalizar e sair do sistema antes de iniciar a piscar, de forma a permitir esse período de espera estabelecido.

O programa *permissions* (C.3) visa testar apenas a chamada *exit*. Nele, nas linhas 8 e 22, é declarado e *derreferenciado* um ponteiro com endereço privilegiado, como mencionado em 4.1. Isso provoca uma exceção de *Data Abort* que acaba chamando a função *forced_exit* criada, que invoca a chamada *exit* no processo infrator com status de falha e acende o LED embarcado na placa, assim expulsando o processo do sistema. Esse processo é visível ao se interromper a sequência do LED piscando (4/8 "piscadas completas"), mas percebe-se que o processo de outro usuário, alheio a este, não é afetado pela exceção provocada.

Além do teste empírico (visual) do programa, fez-se uso extensivo do equipamento de depuração disponível. A Figura 3 apresenta o processo de depuração aplicada com J-Tag e J-Link para o programa *permissions*.

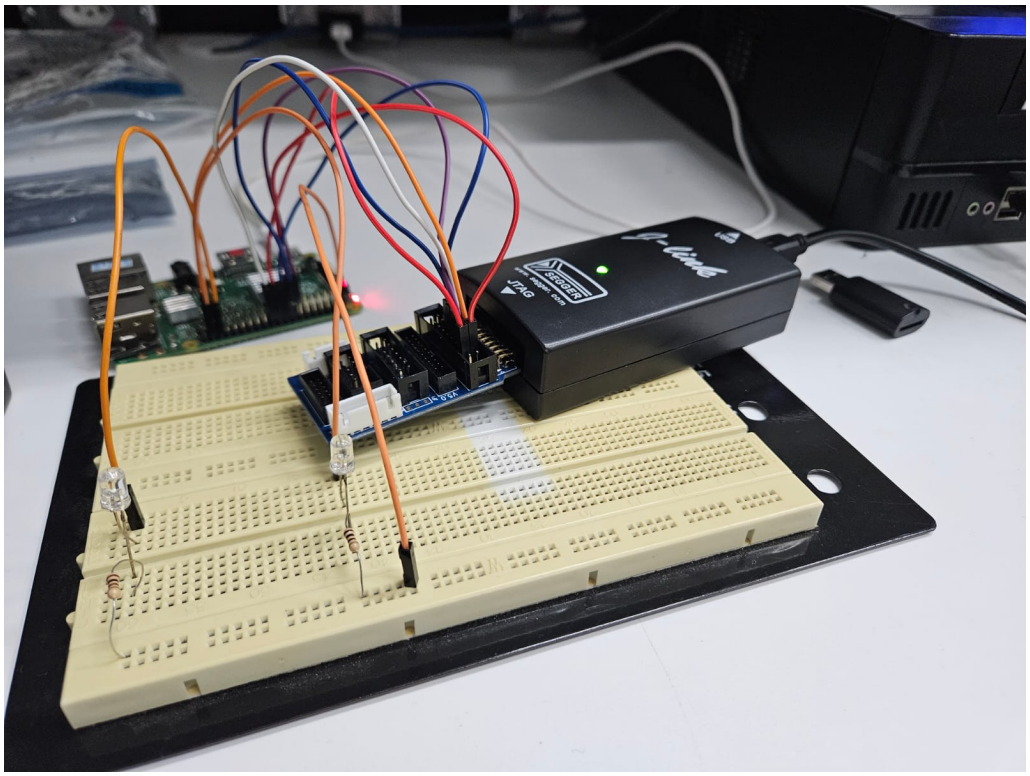

Figura 3: Depuração do programa *permissions.c* através do J-Link com GDB *server*.

6 CONSIDERAÇÕES FINAIS

O *PiFork* é um sistema simples, porém indicativo do gerenciamento complexo de um SO completo. A aplicação de recursos além da unidade "básica"de computação de um processador, envolvendo a configuração do sistema (cache, MMU e TLB) e a integração de modos de operação no tratamento de exceções específicas do sistema permitiu desenvolver uma solução completa - aplicável e estendível.

Ainda que o projeto funcione como proposto, alguns pontos não foram explorados tanto quanto poderiam. A configuração das seções de memória poderia ser ajustável com variáveis de ambiente no momento de compilação, adicionando também o segundo nível da tabela de tradução de endereços. Nesse mesmo contexto, poderia se optar por reservar um espaço de memória extra para cada processo manter sua tabela de tradução, de forma a mudar o valor do registrador $TTBR$ ao invés de remapear o endereço ocupado.

De forma análoga, a configuração da cache poderia ser mais explorada e testada: cache unificada, de instruções e dados; mapeamento completamente virtual ou completamente físico.

Os resultados obtidos são satisfatórios e o projeto cumpriu com seus objetivos propostos.

APÊNDICES

A Chamadas de sistema do *PiFork*

```
#define wait(status)  waitpid(-1, status)

static inline void __attribute__((always_inline)) yield(void) {
  asm volatile("mov r0, #1 \n\t"
               "swi #0     \n\t"
               :
               :
               : "r0");
}

static inline pid_t __attribute__((always_inline)) getpid(void) {
  pid_t pid;
  asm volatile("mov r0, #2 \n\t"
               "swi #0     \n\t"
               "mov %0, r0 \n\t"
               : "=r"(pid)
               :
               : "r0");
  return pid;
}

static inline unsigned __attribute__((always_inline)) getticks(void) {
  unsigned ticks;
  asm volatile("mov r0, #3 \n\t"
               "swi #0     \n\t"
               "mov %0, r0 \n\t"
               : "=r"(ticks)
               :
               : "r0");
  return ticks;
}

static inline pid_t __attribute__((always_inline)) fork(void) {
  pid_t pid;
  asm volatile("mov r0, #4 \n\t"
               "swi #0     \n\t"
               "mov %0, r0 \n\t"
               : "=r"(pid)
               :
               : "r0");
  return pid;
}

static inline pid_t __attribute__((always_inline)) waitpid(pid_t pid,
                                                           int *status) {
  pid_t awaited_pid;
  asm volatile("mov r1, %1 \n\t"
               "mov r2, %2 \n\t"
               "mov r0, #5 \n\t"
               "swi #0     \n\t"
               "mov %0, r0 \n\t"
               : "=r"(awaited_pid)
               : "r"(pid), "r"(status)
               : "r0", "r1", "r2");
  return awaited_pid;
}

static inline void __attribute__((always_inline)) exit(int status) {
```

```
59    asm volatile("mov r1, %0 \n\t"
60                 "mov r0, #6 \n\t"
61                 "swi #0     \n\t"
62                 :
63                 : "r"(status)
64                 : "r0", "r1");
65 }
```

B Cópia de Seção

```
/**
 * @brief Copies a section of 1 MiB pointed by src to dest
 *
 * @param src Page source address
 * @param dest Page destination address
 */
.global section_copy
section_copy:
  push {r4-r10}
  // 2^20 bytes <=> 2^18 words <=> 2^13 iterations
  // (4 instructions, 8 registers each)
  mov r2, #(1 << 13)

copy:
  // Loop Unrolling
  ldmia r0!, {r3-r10}
  stmia r1!, {r3-r10}
  ldmia r0!, {r3-r10}
  stmia r1!, {r3-r10}
  ldmia r0!, {r3-r10}
  stmia r1!, {r3-r10}
  ldmia r0!, {r3-r10}
  stmia r1!, {r3-r10}

  subs r2, r2, #1
  bne copy

  pop {r4-r10}
  mov pc, lr
```

C Programas de Teste

Os programas em C testados (com e sem J-Tag).

C.1 wait

```c
#include <pifork.h>

#define DELAY(instr)  for (int i = 0; i < (instr); i++)  \
                        asm volatile("nop");

void __attribute__((section(".user1"))) user1_main(void) {
  int stat;
  pid_t pid, pid_awaited;

  // Setup GPIO 16, 17 and 19 as output
  GPIO_REG(gpfsel[1]) =
      (GPIO_REG(gpfsel[1]) & (~(0x7 << 18 | 0x7 << 21 | 0x7 << 27))) |
      __bit(18) | __bit(21) | __bit(27);

  pid = fork();
  if (!pid) {
    for (int i = 0; i < 12; i++) {
      GPIO_REG(gpset[0]) = __bit(19);
      DELAY(2000000);
      GPIO_REG(gpclr[0]) = __bit(19);
      DELAY(2000000);
    }
  } else {
    for (int i = 0; i < 8; i++) {
      GPIO_REG(gpset[0]) = __bit(16);
      DELAY(1000000);
      GPIO_REG(gpclr[0]) = __bit(16);
      DELAY(1000000);
    }
    pid_awaited = wait(&stat);

    if (WIFEXITED(stat) && !WEXITSTATUS(stat) && pid_awaited == pid) {
      for (int i = 0; i < pid_awaited; i++) {
        GPIO_REG(gpset[0]) = __bit(17);
        DELAY(500000);
        GPIO_REG(gpclr[0]) = __bit(17);
        DELAY(500000);
      }
    }
  }
  exit(EXIT_SUCCESS);
}

void __attribute__((section(".user2"))) user2_main(void) {
  // Setup GPIO 20 as output
  GPIO_REG(gpfsel[2]) = (GPIO_REG(gpfsel[2]) & ~0x7) | __bit(0);

  while (1) {
    GPIO_REG(gpset[0]) = __bit(20);
    DELAY(600000);
    GPIO_REG(gpclr[0]) = __bit(20);
    DELAY(600000);
  }
  exit(EXIT_SUCCESS);
}
```

C.2 waitpid

```c
#include <pifork.h>

#define DELAY(instr)  for (int i = 0; i < (instr); i++)  \
                        asm volatile("nop");

void __attribute__((section(".user1"))) user1_main(void) {
  int pin;
  pid_t pid = ~0;

  // Setup GPIO 16-19 as output
  GPIO_REG(gpfsel[1]) = (GPIO_REG(gpfsel[1]) &
                         (~(0x7 << 18 | 0x7 << 21 | 0x7 << 24 | 0x7 << 27))) |
                        __bit(18) | __bit(21) | __bit(24) | __bit(27);

  for (int i = 0; i < 3; i++) {
    if (pid) {
      pin = 16 + i;
      pid = fork();
    }
  }

  if (pid) {
    pin++;
    waitpid(pid, NULL);
  }

  for (int i = 0; i < 8; i++) {
    GPIO_REG(gpset[0]) = __bit(pin);
    DELAY(600000);
    GPIO_REG(gpclr[0]) = __bit(pin);
    DELAY(600000);
  }

  exit(EXIT_SUCCESS);
}

void __attribute__((section(".user2"))) user2_main(void) {
  while (1)
    ;
  exit(EXIT_SUCCESS);
}
```

C.3 permissions

```c
#include <pifork.h>

#define DELAY(instr)  for (int i = 0; i < (instr); i++)  \
                        asm volatile("nop");

void __attribute__((section(".user1"))) user1_main(void) {
  // Maps to the first 1 MiB section (kernel)
  volatile int *ptr = (int *)0xC0DE;

  // Setup GPIO 16 as output
  GPIO_REG(gpfsel[1]) = (GPIO_REG(gpfsel[1]) & ~(0x7 << 18)) | __bit(18);

  for (int i = 1; i <= 8; i++) {
    GPIO_REG(gpset[0]) = __bit(16);
    DELAY(2000000);
    GPIO_REG(gpclr[0]) = __bit(16);
    DELAY(2000000);
```

```
18
19      if (i == 4) {
20        // Invalid data access (privileged region)
21        // provokes forced exit
22        *ptr = 69;
23      }
24    }
25
26    exit(EXIT_SUCCESS);
27  }
28
29  void __attribute__((section(".user2"))) user2_main(void) {
30    // Setup GPIO 20 as output
31    GPIO_REG(gpfsel[2]) = (GPIO_REG(gpfsel[2]) & ~0x7) | __bit(0);
32
33    while (1) {
34      GPIO_REG(gpset[0]) = __bit(20);
35      DELAY(600000);
36      GPIO_REG(gpclr[0]) = __bit(20);
37      DELAY(600000);
38    }
39    exit(EXIT_SUCCESS);
40  }
```